Pesto Flavoured Security

Feike W. Dillema and Tage Stabell-Kulø

Technical Report 2002-42

May 2002

Abstract

Pesto aims at providing highly available and secure storage for long-lived data to mobile users roaming into untrusted environments.

Security in Pesto encompasses the following three aspects: availability, safety, and privacy. A mechanism supporting one aspect may adversely affect another. For example, replication may increase availability but complicates supporting confidentiality, and simply encrypting data for confidentiality may defeat the whole purpose of replication. We show that an integral approach to these aspects leads to considerable savings in overall system complexity, and thus to a more secure system.

In Pesto, users may specify different levels of trust in different parts of the infrastructure. In particular, a user may trust a node to merely store (encrypted) data, and/or to distribute replicas to other nodes on his behalf, and/or he may trust a node to enforce access control on his behalf to his (plaintext) content. This report gives an overview of the main security mechanisms that makes this separation of concerns possible. We present its novel encryption framework and its trust management and discuss how it can be used to build distributed infrastructures with advanced security and safety properties.

1 Introduction

When users embrace mobile computing, they typically end up with multiple machines, some stationary and some mobile. As a consequence, they have a need to share data between these machines. Mobile computing adds the requirement that users should be able to roam between many different environments. Independent of what administrative domain they roam into, they want access to their personal resources while using resources available in that domain. As the content and the resources being shared may then belong to different administrative domains, there is a need to separate the mechanisms for sharing content and resources. Furthermore, users want to share data with other users. Such sharing should be possible between any two communicating users, independent of their current environment and connectivity to other parts of the infrastructure.

We believe that for a system to be viable in a rapidly diversifying world, the overall system design can not rely on a common policy being feasible. Not

a common policy for authentication, not for authorization, and not for the cost of resources. Furthermore, we believe that the users should be provided with the means to establish any relationship to service providers as they wish; it is the goal of Pesto to utilize the relationships the user might have.

The Pesto system aims at providing an infrastructure that meets the needs of mobile users for distributed, reliable and secure storage. We believe that the crux of the matter is related to trust, and management of trust relations. An important part of Pesto is to separate trust relations from administrative relations, and to separate availability of *storage resources* from accessibility of *content*.

Pesto consists of nodes that communicate with each other using an asynchronous request-response protocol that supports two types of requests; one to fetch data from a node, and one to store data at a node. A Pesto node is owned by a user who has authority over its local storage resources. A user may acquire the right to use storage resources at remote nodes by means of a service contract, and he may delegate rights to use his resources to others. All data stored in Pesto is encrypted with a shared encryption key. Access to storage space, i.e. fetching and storing encrypted data at a node, is thus separated from access to content, i.e. access to the encryption key needed to decrypt that data. Hence, Pesto nodes can share storage resources independently from actual content.

The collection of nodes in a Pesto system do not offer its users a single global system view with pre-defined relationships between (subsets of) nodes. This means, for example, that there is no system-wide trusted computing base. In general, no special sets of nodes are defined, a priori, to be more reliable, safer, more secure or more capable of performing any specific task on behalf of a user. Pesto allows the user to define what nodes may execute specific tasks on his behalf, based on his personal requirements, beliefs and current needs. The design of Pesto carefully separates the different mechanisms a distributed storage infrastructure must support. Because of Pesto's design, a user can place responsibility of the different tasks on different sets of machines, possibly governed by a variety of administrative domains. In this way, the user is free to set up relationships he is comfortable with and assign tasks to different parts of the infrastructure accordingly.

2 Files

The Pesto storage system provides distributed storage of files to its users. Pesto stores and replicates the complete update history of files. More precisely, a file in Pesto is an initial (empty) version of it, together with all updates made to that file. As such, any version of a file can be retrieved at any time. An application may choose to store the actual differences between the current version and a new one, or it may store the complete new content as a file update. Actually, Pesto puts no constraints on the content of a file update at all, and leaves it to applications to define their own update and access semantics.

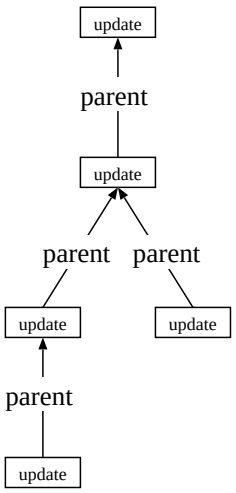

Figure 1: a file

A file update is immutable and is identified by a *glob-*

ally unique identifier (GUID) which is a 128-bit random number. Due to the random selection from an immense name space, a new GUID can be generated locally with no risk of an identical name existing anywhere in the system.

Each file update keeps a reference to the previous update to the file, its parent update, such that a file is organized as a file update tree. A file consists therefor of an ordered set of identifiable updates. An example file is shown in Fig. 1. The combination of replication and independent, concurrent updates requires that the issue of consistency must be considered. In general, it is impossible to ensure that no concurrent updates occur, unless one is willing to accept that an update blocks until the global state of the system can be determined. However, since each update in Pesto has an unique name, the two (or more) concurrent updates are identifiable. Because they are concurrent, they will have the same parent, e.g. they are derived from the same version. Hence, the ordered set of updates that represents a file is actually a tree, and a file is thus the set of updates, possibly organized as a tree, that together constitutes its content.

Because each and every update is uniquely named, any version of a file can be retrieved. The details of how the application chooses to present this fact to the user does not concern Pesto. Furthermore, we believe that it should be left to each and every application, and ultimately to the user, how to deal with a concurrent update (a branch in the version tree). This design-choice implies that the term replication in the context of Pesto merely refers to the activity of distributing copies of file updates to a user-specified set of nodes. It does in particular not include consistency control. Pesto thus separates replication from consistency control, and the storage system itself only provides replication; the advantage is a significant simplification in the design.

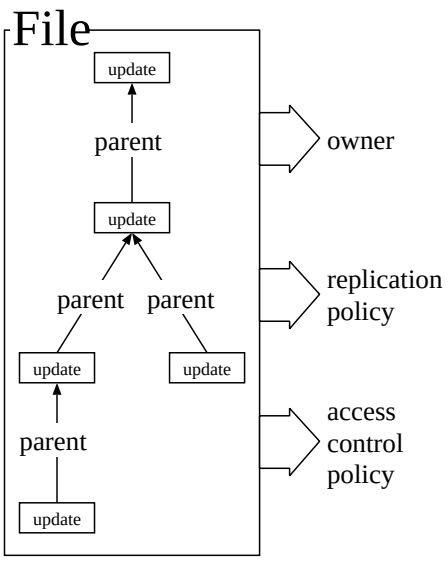

Figure 2: file references

A replication policy, an access control policy and an owner are associated with a file, as shown in Fig. 2 for the example file. A Pesto node maintains a special file that contains a variety of administrative information about it. This file describes and represents the node, its owner and its storage resources. Its GUID is used to identify and address the node in the system. A file references its owner by the GUID of this file.

Policies are also stored as files. A file references a policy that applies to it, by the GUID of the file that stores the policy. These policy references are specified by the user at file creation time and are immutable. A given file can thus never be associated with a policy identified with another GUID. The content of the policy files themselves can be changed, of course.

The owner of a file is the Pesto node that created it. The GUID of the owner

is kept with the file and is immutable. This means that ownership of a file is not transferable from one user to another, other than by making a copy of the file that has a new GUID. Actually, each file update has a creator associated with it. This creator is the node that authorized its creation. The owner of a file is then the same as the creator of its first version/update.

A replication policy specifies the set of nodes the user expects to store a replica of his file, and it specifies the nodes responsible for the distribution of the replicas to these nodes (notice the difference). Distribution of replicas according user specified policies proceeds whenever communication between source and destination node is possible, i.e. is independent of any synchronization with other replicating nodes. As there is no mechanism (protocol request) for deleting individual files or file updates by GUID, the only way to ask a node to remove files from its local storage is by removing that node from a replication policy. Note, however, that such a request will apply to all files that are governed by that replication policy.

An access control policy specifies what credentials a user deems sufficient for a request to be granted access to the content encoded by his file (i.e. who is allowed access to its updates). As the storage resources of a node are described and represented by a file, a regular access control policy can be used to specify who should be granted access to these storage resources. In other words, access to content and storage resources are separated, but governed by the same mechanisms.

In short, we can say that Pesto stores the complete update history of content in files, and it stores all state of the system in such files. References by GUID are used to associate files and policies with each other.

3 Security

Pesto makes it possible for users to implement any security policy they choose. In addition, Pesto allows the user to specify who else is trusted to enforce a security policy that he has defined and thus should be granted the ability to do so. Pesto itself is designed around a very simple base security policy: all communication and all stored content is regarded confidential and access is only granted to the user that created it. In other words, data not properly encrypted is not accepted by the storage system, and the encryption keys are initially only available to the user. A user may relax this base security policy for the files he creates by specifying what other Pesto nodes should be granted access to the various encryption keys in use. Pesto is responsible for securely distributing the encryption keys to nodes that the user trusts.

The main application of encryption in Pesto is to create a separation between access to storage (of encrypted data) and access to content (via encryption keys). The encryption framework facilitates (and separates) access control for reads and updates. Using the mechanisms provided by Pesto, it is possible to delegate authority over access control decisions to others.

3.1 Encryption

The complete update history tree is stored with every file, as a set of *file updates*. The content of each file update is encrypted with a different encryption key.

A key used to encrypt a single file update is called a *member key*. Read-access control is then exercised by controlling who has access to a file's member keys. The member keys of a file are subsequently encrypted with the so-called *file key*.

The example file depicted in Fig. 1 thus uses five member keys and a single file key as shown in Fig. 3. Each encrypted member key is stored as part of the file update it belongs to. The file key is generated when the file is created, at which time it is made available to the owner of the file. Because of the way keys are managed, a user who knows the file key is able to get hold of all member keys, which gives him read access to all file updates (e.g. all versions of the content). Obviously, by handing out a single member key, read access is granted for individual file updates.

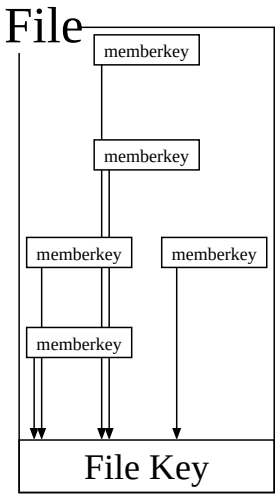

Figure 3: file encryptions

3.2 Update Authorization

Update access (writes) to files is controlled by the same mechanism used to control read access. When a request to update a file is received in the form of a new file update, it is considered authorized if the update is encrypted with a fresh member key, and that key is found encrypted with the file key for that file. In other words, update access can only be granted by someone that knows the file key.

A member key is deemed fresh when no (locally) existing file update of its file is encrypted with that member key. A node that knows the file key can check the encryption of the file update and the freshness of the member key used, and decide whether the update is properly authorized.

Anyone who controls the file, i.e. has access to the file key, can properly encrypt a new member key and hand it to any user (including himself). This constitutes a delegation of authority over the file for the purpose of a single update. Notice how this mechanism ensures that a user can be given authority to update a file one single time, without him learning the file key. Hence, the authority is not reusable and can not be delegated further.

3.3 Revocation

As a consequence of how authority over updates is implemented, authorization to make a file update can be separated in time from the actual injection of the update into the storage system. There may then sometimes be a need to revoke this authorization after it has been granted but before it is used. Imagine, for example, situations where users hoard authorizations or try to save authorizations for use long after the original credentials used to acquire them expired. A user may find it useful or required to avoid such situations. In order to revoke authorizations, the user that issued the authorization can store an 'empty' file update with the GUID and member key in question himself. Due to the WORM (Write Once Read Many times) access semantics of file up-

dates stored by Pesto, this effectively renders the authorization harmless and so revokes it.

3.4 Granularity of Access

The encryption framework presented so far achieves that access control can be exercised at fine granularity. Both read and update access is based on access to member keys (respectively to existing and newly created member keys). That is, the unit of access is the file *update*. As described in Sec. 2, there are no constraints on what the content of a file update actually constitutes. It is left to the application to specify file update semantics. This also means that applications can determine the unit of access control, i.e. the granularity of access control.

A straightforward example is a versioning file system application, that stores each file version as a Pesto file update, such that the unit of access control is a file version and access to different versions can be controlled independently of each other. Pesto itself defines somewhat less conventional update semantics for its administrative and policy files, with as main goal enforcement of access control at a fine enough granularity. These semantics are largely outside the scope of this paper, but we want to give a hint of what is possible here using replication policy files as example.

As described earlier, a replication policy file includes the user-specified list of replicating nodes. Pesto actually stores each member of this list as a separate file update to the replication policy file. This way, an access control policy to this replication policy can be enforced that, for example, prevents one replicating node to find out about the location of other replicas.

3.5 Access Control Policy

We described in Sec. 2 how each file references an access control policy by the GUID of the file that stores it. Such an association between a file and its access control policy needs to be protected by cryptographic means in order to be useful. To that end, an access control policy file contains two encryption keys. These keys are called the *read access key* and the *update access key* respectively. The read access key of an access control policy is used to encrypt the member keys of the files that are governed by the policy. The update access key of an access control policy is used to encrypt the file key of each file that references the policy.

A member key encrypted with the read access key is stored and distributed with its file update. A file key encrypted with the update access key is stored and distributed with its file. The example file depicted in Fig. 1 thus stores five member keys encrypted with the file key, five member keys encrypted with the read access key, and one file key encrypted with the update access key of its access control policy as shown in Fig. 4.

Basically, we group files under their respective access control policy. We avoid storing and distributing a potentially very large number of member keys and file keys with the policy. This is achieved by adding an extra level of cryptographic indirection, i.e. two new keys that encrypt those member keys and file keys instead. The result is that the (encrypted) member keys and file keys can simply be replicated together with the files, to increase safety and availability.

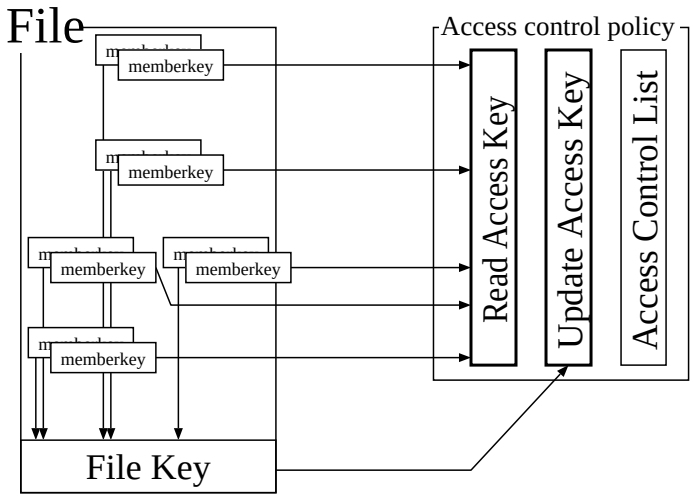

Figure 4: file encryption

Of course, the requirements for secrecy, safety and availability have moved from the member keys and file keys to the access keys, i.e. we still need to keep these two keys safe and secure. However, this is a much simpler problem to solve. Not only are there far fewer keys to protect, but more important is that the qualities of these two keys are substantially different from those of the set of member and file keys. In particular, while the set of member and file keys will grow over time as new files are created, the set of two access keys does not change with the number of files they apply to.

As only nodes that hand out member keys to users need to know the access keys, it will typically be easier to maintain their secrecy. Safety is also easier to maintain due to the small amount of data involved and because this data is immutable. It will typically be feasible to replicate the access keys to physically secure and safe (offline) media, like a smartcard kept in a safety deposit box of a bank. To achieve availability, we expect the mobile user to carry his access keys with him on a smartcard or small handheld computer, optionally protected with a passphrase in addition. Hence, secrecy is achieved without complicating the implementation of safety and availability requirements.

4 Trust

The design of Pesto carefully separates the different mechanisms a distributed storage infrastructure must support. Because of this, a user can place responsibility of the different tasks on different machines (nodes), possibly residing in a variety of administrative domains. The user is then free to set up relationships he is comfortable with and assign tasks to different parts of the infrastructure. The trust relation a user has with the (owner/manager of the) node will determine how the user will use the node in his Pesto configuration. Note that there is really no such thing as a “Pesto system”. There is merely a collection

of completely independent Pesto nodes, and each and every user determines to what extent he trusts a particular node.

A Pesto node can be assigned one or more rôles by any particular user. A node can be assigned the responsibilities for the tasks of storing files, distributing files and/or enforcing access control to files on behalf of the user. The manner in which encryption is applied in Pesto, reduces assignment of such a rôle to a node to the exchange of an encryption key. An assignment of responsibility is limited to a user-specified set of files. In other words, different nodes may be assigned responsibilities regarding different sets of files. As a result, a user can specify three different sets of nodes for each of his files. These sets are called the trusted storage base, the trusted replicator base, and the trusted access base, respectively.

4.1 Trusted Storage Base

The trusted storage base (TSB) of a file is the set of nodes the user trusts to store an (encrypted) replica of that file. A member of a TSB is only trusted to store data, it is not trusted with the keys that protect the data. Nodes in a TSB are thus 'merely' storage providers for the user and they perform access control to their storage resources, not of the content they store.

Users negotiate service contracts with other users in any way they deem appropriate. We envision that users establish service contracts with other users in a variety of ways; online storage service providers offering services for a fee, cooperative users exchange storage for storage, and companies might offer their employees access to its storage resources. A service contract could take many different forms, from signed paper contracts to electronic contracts or verbal agreements, to name just a few. How such contracts are established is of no concern to Pesto, regardless of the relationship and conditions of use.

Pesto's only requirement on a service contract negotiation is that somehow a secret (key) is exchanged as part of it. This secret is subsequently used to authorize requests to use the negotiated storage resources. Therefor, a user shares a secret key with each of his storage providers. A request to use the negotiated storage resources (i.e. store a file update) is considered authorized if the request is properly encrypted with this shared key, and accounting by the storage provider (if applied) shows enough negotiated storage resources are still unused by the user.

If a user wants to act as storage provider for other users, he does so by specifying an access control policy for his storage resources. How this is implemented is outside the scope of this paper. We only note that access to storage resources uses the same mechanisms as access to content. Actually, access control of storage resources is reduced to access control of the file that describes them. This assists in reducing the complexity of the storage system.

4.2 Trusted Replicator Base

The trusted replicator base (TRB) of a file are the nodes trusted to enforce its replication policy on behalf of the user; that is, each TRB member is responsible for the *distribution of replicas* to some subset of nodes in the trusted storage base of the file. Together with the TSB, the nodes in a TRB make up a directed distribution graph with edges leaving only nodes in the TRB and ending in

nodes in the TSB. A user shares a secret key with each of his replicators, like he does with his storage providing nodes.

In order to perform their assigned task, a member of a TRB needs authority to use storage resources negotiated by the user at the nodes it is expected to distribute file updates to. A straightforward implementation would support this by handing the secret key shared between the user and the storage provider to the replicator node. In Pesto, however, we delegate authority from this key to a new key which is subsequently installed both at the storage provider and the replicator. This facilitates easy revocation of such a delegation when a user removes an individual node from a TRB.

4.3 Trusted Access Base

The trusted access base (TAB) of a file are the nodes that are trusted to enforce an access control policy on behalf of the user. Actually, two separate trusted access bases can be specified for a file; one for read-only access one for read/update access. Members of the former are trusted to handout read access only, and are handed the read access key in order to enable them to do so. The members of the latter are trusted to perform both read and update access control and are handed the update access key associated with the access control policy in question.

Pesto itself is not responsible for enforcing these user-specified access control policies, and thus does not prescribe its format and contents. In other words, the user (with help of his management applications) may specify what (type of) credentials he finds necessary and sufficient to authorize access of some kind. The storage system is merely responsible for distributing such policies, together with the encryption keys required to enforce them, to the relevant trusted access bases.

4.4 Responsibility and Risk Management

Pesto keeps different responsibilities separate, so that the user can allocate them to different, but possibly overlapping, parts of the infrastructure. Other responsibilities than presented here could be defined. For example, consistency control and a 'trusted consistency base' could be defined accordingly. As consistency requirements are highly application dependent, Pesto only provides a basic synchronization mechanism that applications can use to construct their own consistency protocols. It is outside the scope of this paper to discuss consistency control and ways to enhance the security of such protocols (see e.g. [4, 7]).

As described, authority is delegated from the user to an encryption key. This reduces the assignment of responsibilities to a key management problem. By the *scope* of a key, we mean the information that is available to a user that knows the key. For example, the scope of a member key is thus a single file update, and it has a very limited scope compared to an update access key. An important aspect of our encryption framework is that each key has a well-defined and limited scope. This assists the user in assessing the risks of using potentially untrustworthy machines in order to make progress. Pesto is not designed to deny its users service in cases where a user deems it more important

to continue working than to protect the confidentiality of that work. Instead, Pesto is designed to limit the risk involved in such actions.

5 Discussion

Our design has a user-centric view in that all authority in the system originates from individual users and not from inside the system itself. The access control and replication mechanisms do not mandate any hierarchical, fixed or static structure on administrative domains. This makes Pesto suitable for building personal ad-hoc infrastructures for sharing between individuals, but it is certainly not limited to such.

Individual administrative domains can be used as building block to construct larger domains using delegation. This requires cooperation from the individual users as they cannot be forced by the system to delegate their authority to others. Enforcement of mandatory policies would require all nodes to be under full control of a single authority. But, even then, users may evade the mandated controls using channels outside the system. Extending the reach of the mandated policy to include such channels as telephone and floppy-disks is infeasible in all but the most restricted environments (like intelligence agencies, the military, criminal organizations).

We believe that the lack of mandatory transfer of authority is not a weakness in our design, but reflects that in most real-world environments, user cooperation is a requirement to enforce a shared policy. Cooperation from non-malicious users may possibly be 'bought' instead, by making the use of shared resources conditional to such cooperation. For example, a company could define a storage policy for its file servers that allows only storage of files of employees, for which it has been delegated the rights to define the TAB, TRB and/or the TSB. In other words, the file servers will only store files that reference policies owned by the company. It could, in addition, set up its communication infrastructure such that only the servers under its control may be reached through it.

5.1 Denial of Service

Our design is well suited for the construction of publication infrastructures similar to "The Eternity Service" [1] and "The XenoService" [9], that are quite resilient to denial of service attacks.

Denial of service is targeted at destroying a certain resource or at exhausting the resources of the service providers. Replication of files together with logging of all file updates assists in preventing the former. Resilience to resource exhaustion attacks can be gained by protecting resource use and/or by ensuring more resources are available during the attack than an attacker is able to consume. Resource protection is supported by separating access to storage from access to content and the asynchronous nature of the protocol used. Access decision for content only need to be taken into consideration after access to storage use has been granted. When the system is flooded with requests for storage resources, one can simply ignore all these without denying service to users that were granted access to storage earlier. Graceful degradation during

periods of semi or full disconnected operation limits the damage of a successful network denial of service attack.

The ability to turn secret members of a TSB into members of a public TAB with merely the exchange of a single encryption key, can be used to increase the amount of available resources dynamically during an attack to counter the resources consumption of the attack.

5.2 Key Management

Pesto has been designed around our so-called '*Open-End Argument*' design guideline [8]. According to it, *the user* should be solely in charge of important matters such as how identity of users is represented and checked and what kind of credentials are needed to authorize actions. Pesto does not support any notion of authentication of "users".

By leaving to the user to decide what credentials are considered "good enough", Pesto does not prescribe the structure and organization of the user community. For example, a "web of trust" structure constructed with PGP [10] could be used as well as some kind of certification authority hierarchy. In other words, the user is free to use any existing infrastructure to realize any means of authentication and authorization he might fancy.

We use a SPKI based infrastructure 'on top of' Pesto, as basis for more advanced authentication, authorization and access control [2, 3, 5]. This infrastructure is structured as a client-server architecture. While its complete implementation consists of about 30.000 lines of Java code, clients have been written that are small and efficient enough to run on a small hand-held device, like a Palm Pilot [6].

Because SPKI is very flexible indeed, sophisticated certificate-based 'services' such as on-line verification, revocation, and once-only semantics can be offered 'on top of' Pesto. A user performs authentication and authorization based on chains of SPKI certificates and uses SPKI certificates to construct his access control policy. Of course, the user will often delegate enforcement of such a policy to an application that speaks on his behalf and runs on the nodes of his trusted access bases. Actually, the same application can be used on storage providing nodes to control access to storage. Delegation is, of course, performed by giving the application the relevant access keys that Pesto manages.

6 Conclusions

Pesto is a flexible distributed storage system simple enough to include resource-poor devices of mobile users. It allows its users to specify what part of the infrastructure is trusted to perform each specific task on behalf of the user. It supports incorporation of the personal and business trust relationships into the system, while not dictating or assuming such relationships as part of the design itself. Mechanisms to increase safety and availability have been designed together with privacy protection mechanisms, providing ease of management and resilience against user error and violation of trust.

Our encryption framework makes read and update access control possible without relying on public-key cryptography. Public-key cryptography can,

however, be used for delegation and authorization on top of Pesto's base security mechanisms. A user need only carry a handful of encryption keys with him on his mobile machine in order to be able to off-load work to nearby, better-connected machines. This makes Pesto suitable for networks that are semi-partitioned. Support for acquisition of authorization before actual use of it provides solid and secure support for disconnected operation.

The encryption framework is designed to limit the risks of and support recovery from a user's mistakes, misjudgements and deliberate risky and insecure behaviour. Pesto does not deny its users service in cases where a user deems it more important to continue working than to protect the confidentiality and integrity of that work. Pesto limits the risks involved to the work in question only.

The separation of access to data and content resulted in that safety and privacy can be addressed separately. The degree of replication can be increased to obtain better availability without having to consider the trustworthiness or privacy policy of new storage providers. Aiming at providing good end-to-end security and safety at the same time, resulted in an elegant design were the security and safety mechanisms support each other with the addition of a minimum of complexity.

References

[1] Ross Anderson. The Eternity service. In *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology*, 1996.

[2] Carl M. Ellison. SPKI Requirements. RFC 2692, The Internet Society, September 1999.

[3] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. RFC 2693, The Internet Society, September 1999.

[4] Maurice Herlihy and J. D. Tygar. How to make replicated data secure. In *Proceedings of Advances in Cryptology, CRYPTO '87*, number 293 in Lecture Notes in Computer Science, pages 379–391. Springer Verlag, 1988.

[5] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[6] Per Harald Myrvang. An infrastructure for authentication, authorization and delegation. Cand.scient. thesis, Department of Computer Science, University of Tromsø, Norway, May 2000.

[7] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[8] Tage Stabell-Kulø, Feico Dillema, and Terje Fallmyr. The open-end argument for private computing. In Hans-W. Gellersen, editor, *Proceedings of the ACM First Symposium on Handheld, Ubiquitous Computing*, number 1707 in Lecture Notes in Computer Science, pages 124–136. Springer Verlag, October 1999.

[9] Jeff Yan, Stephen Early, and Ross Anderson. The XenoService - A Distributed Defeat for Distributed Denial of Service. In *ISW 2000, IEEE Computer Society, Boston, USA*, October 2000.

[10] Phillip Zimmermann. *The official PGP user's guide*. The MIT Press, 1995. ISBN 0-262-74017-6.